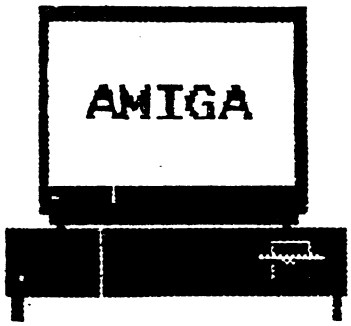

BECAUSE

ISSUE 4
AUGUST 1987

BULLETIN OF THE CANBERRA AMIGA USERS SOCIETY

(The Journal that explains the mysterious
happenings inside your Amiga - BECAUSE)

MEETINGS:

Meetings are held on the second Wednesday of each month on the second floor of the J.G. Crawford building at ANU, starting at 8:00 PM. Roll up any time after 7.30.

The next meeting will be held on the 16th of September.
The following meeting will be on the 21th of October.
***** Annual General Meeting *****
The October meeting will be our 1987 AGM where new committee members are to be elected.

SUBSCRIPTIONS:

Annual fees are $20, payable at any of the monthly meetings to the treasurer.

AIMS/BENEFITS:

CAUSe is an independent group formed to bring together people who own, use or are interested in the Commodore Amiga computer. Members receive a bulletin providing a wealth of information concerning the Amiga and are encouraged to attend our monthly meetings.

COMMITTEE:

Treasurer: John Bishop --- 49 3786 (W) Editor: Craig Fisher - 54 6033 (H)
Secretary: Chris Townley - 41 3209 (H) Auxiliary: Peter McNeil - 54 2732 (H)

BULLETIN BOARD:

CAUSE has it's own FIDO-NET bulletin board which is can be used free of charge to anyone with a modem. There are now two phone lines on the board, one of which is sponsored by CAUSE.

The board is Baud rate auto-sensing and will accept 300/300, 1200/1200, 1200/75 and now also 2400/2400 baud rates. Available 24 hours a day.

Sysop: Mike Hurst-Meyers

BBS: 59 1137 (all speeds) or 59 1440 (except 300 baud CCIT)

IN THIS ISSUE:

RJ Mical at the Boston Computer Society
By Gary Oberbrunner

Reprinted from Amiga Workbench,
newsletter of the Amiga Users Group,
PO Box 48, Boronia, 3155.

On Monday March 2, RJ Mical (=RJ=) spoke at the Boston Computer Society meeting in Cambridge. Fortunately I was momentarily possessed with an organizational passion, and I took copious notes. I present them here filtered only through my memory and my Ann Arbor. My comments are in [square brackets].

What follows is a neutron-star-condensed version of about three and one half hours of completely uninterrupted discussion.

PART 1 - The Rise and Fall of Amiga Computer Inc.
The Early Days

Amiga Computer Inc. had its beginnings, strangely enough, RJ began, with the idea of three Florida doctors who had a spare $7 million to invest.

They thought of opening a department store franchise, but (as RJ said) they wanted to try something a bit more exciting. So they decided to start a computer company. "Yeah, that's it! A computer company! That's the ticket! :-)"

They found Jay Miner, who was then at Atari (boo hiss) and Dave Morse, the VP of sales (you can see their orientation right off) they lifted from Tonka Toys. The idea right from the start was to make the most killer game box they could. That was it, and nothing more. However Jay and techies had other ideas. Fortunately they concealed them well, so the upper management types still thought they were just getting a great game machine. Of course the market for machines like that was hot hot hot in 1982.

They got the name out of the thesaurus; they wanted to convey the thought of friendliness, and Amiga was the first synonym in the list. The fact that it came lexically before Apple didn't hurt any either, said RJ. However before they could get a machine out the door, they wanted to establish a "market presence" which would give them an established name and some distribution channels - keep thinking "game machine" - which they did by selling peripherals and software that they bought the rights to from other vendors. Principal among these was the Joyboard, a sort of joystick that you stand on, and you sway and wiggle your hips to control the switches under the base. They had a ski game of course, and some track and field type games that they sold with this Joyboard. But one game the folks at Amiga Inc. thought up themselves was the Zen Meditation game, where you sat on the Joyboard and tried to remain perfectly motionless. This was perfect relaxation from product development, as well as from the ski game. and in fact, this is where the term Guru Meditation comes from; the only way to keep sane when your machine crashes all the time is the ol' Joyboard. The execs tried to get them to take out the Guru, but the early developers, bless 'em, raised such a hue and cry they had to put it back in right away.

When RJ interviewed with Amiga Computer (he had been at Williams) in July 1983, the retail price target for the Amiga was $400. Perfect for a killer game machine. By the time he accepted three weeks later, the target was up to $600 and rising fast. Partly this was due to the bottom dropping completely out of the game market; the doctors and the execs knew they had to have something more than just another game box to survive. That's when the techies' foresight in designing everything from disk controllers to keyboard (yes, the original original Amiga had NO KEYBOARD), ports, and disk drives began to pay off.

The exciting part of the Amiga's development, in a way its adolescence, that magical time of loss of innocence and exposure to the beauties and cruelties of the real world, began as plans were make to introduce it, secretly of course, at the winter CES on January 4th, 1984(?).

Adolescence

The software was done ten days before the CES, and running fine on the simulators. Unfortunately when the hardware was finally powered up several days later, (surprise) it didn't match its simulations. This hardware, of course, was still not in silicon. The custom chips were in fact large breadboards, placed vertically around a central core and wired together round the edges like a Cray. Each of the three custom "chips" had one of these towers, each one a mass of wires.

According to RJ, the path leading up the first Amiga breadboard, with its roll-out antistatic flooring, the antistatic walls just wide enough apart for one person to fit through and all the signs saying Ground Thyself, made one think of nothing so much as an altar to some technology god.

After working feverishly right up to the opening of the CES, including most everybody working on Christmas, they had a working Amiga, still in breadboard, at the show in the booth in a special enclosed gray room, so they could give private demos. Unfortunately if you rode up the exhibit-hall escalator and craned your neck, you could see into the room from the top.

The Amiga was, RJ reminisced, the hardest he or most anyone there had ever worked. "We worked with a great passion. My most cherished memory is how much we cared about what we were doing. We had something to prove - a real love for it. We created our own sense of family out there."

After the first successful night of the CES, all the marketing guys got dollar signs in their eyes because the Amiga made SUCH a splash even though they were trying to keep it "secret." And so they took out all the technical staff for Italian food, everyone got drunk and then they wandered back to the exhibit hall to work some more on demos, quick bug fixes, features that didn't work, and so on. At CES everyone worked about 20 hours a day, when they weren't eating or sleeping.

RJ and Dale Luck were known as the "dancing fools" around the office because they'd play really loud music and dance around during compiles to stay awake. Late that night, in their drunken stupor, Dale and RJ put the finishing touches on what would become the canonical Amiga demo, Boing. At last the true story is told.

Money Problems

After the CES, Amiga Inc. was very nearly broke and heavily in debt. It had cost quite a bit more than the original $7 million to bring the Amiga even that far, and lots more time and money were needed to bring it to the market. Unfortunately the doctors wanted out, and wouldn't invest any more.
So outside funding was needed, and quick.

The VP of Finance balanced things for a little while, and even though they were $11 million in the hole they managed to pay off the longest-standing debts and keep one step ahead of Chapter 11. After much scrounging, they got enough money to take them to the June CES; for that they had REAL WORKING SILICON. People kept peeking under the skirts of the booth tables asking "Where's the REAL computer generating these displays?" Now money started flowing and interest was really being generated in the media. And like most small companies, as soon as the money came in the door it was spent. More people were added - hardware folks to optimize and cost-reduce the design; software people to finish the OS. Even the sudden influx of cash was only enough to keep them out of bankruptcy, though; they were still broke and getting broker all the time.

How much WOULD have been enough? RJ said that if he were starting over, he'd need about $49 million to take the machine from design idea to market. Of course Amiga Inc. had nowhere near that much, and they were feeling the crunch. Everybody tightened their belts and persevered somehow. They actually were at one point so broke they couldn't meet their payroll; Dave Morse, the VP of Sales, took out a second mortgage on his house to help cover it, but it still wasn't enough.

They know they were going under, and unless they could find someone quick to buy them out they were going to be looking for jobs very shortly. They talked to Sony, to Apple, to Phillips and HP, Silicon Graphics (who just wanted the chips) and even Sears. Finally they called Atari. (Boo! Hiss! [literally - the audience hissed at Jack Tramiel's name!]) Trying to be discreet, RJ's only personal comment on Jack Tramiel was (and it took him a while to formulate this sentence) "an interesting product of the capitalist system." Ahem. Apparently Tramiel has been quoted as saying "Business is War." Tramiel had recently left Commodore in a huff and bought Atari "undercover" so that by the time he left C= he was already CEO of Atari.

Realizing that Commodore was coming out with their own hot game machine, Tramiel figured he'd revenge himself on them for dumping him by buying Amiga Inc. and driving C= down the tubes with "his" superior product. So Atari gave them half a million just for negotiating for a month; that money was gone in a day.

Of course Tramiel saw that Amiga Inc. wasn't in a very good bargaining position; basically unless they were bought they were on the street. So he offered them 98 cents a share; Dave Morse held out for $2.00. But instead of bargaining in good faith, every time Morse and Amiga tried to meet them halfway their bid went down!

"Okay, $1.50 a share.
No, we think we'll give you 80 cents.
"How about $1.25?
70 cents."
And so on...

Even Dave Morse, the staunchest believer in the concept that was the Amiga, the guiding light who made everyone's hair stand on end when he walked into the room, was getting depressed. Gloom set in. Things looked grim.

Then, just three days before the month deadline was up, Commodore called. Two days later they bought Amiga Inc. for $4.25 a share. They offered them $4.00, but Dave Morse TURNED THEM DOWN saying it wasn't acceptable to his employees; he was on the verge of walking out when they offered $4.25. He signed right then and there.

The Commodore Years

Commodore gave them $27 million for development; they'd never seen that much money in one place before. They went right out and bought a Sun workstation for every software person, with Ethernet and disk servers and everything. The excitement was back.

Commodore did many good things for the Amiga; not only did they cost-reduce it without losing much functionality, they had this concept of it as a business machine; this was a very different attitude from what Amiga Inc. had been working with. Because of that philosophy, they improved the keyboard and make lots of other little improvements that RJ didn't elaborate on.

What could Commodore have given them that they didn't? The one thing RJ wanted most from them was an extra 18 months of development time. Unfortunately Commodore wasn't exactly rich right then either, so they had to bring out the product ASAP [and when is it ever any different?] Also, he said, they could have MARKETED it (applause!). If he'd had that extra 18 months, he could have made Intuition a device rather than a separate kind of thing; he could have released it much more bug-free. As far as marketing goes, the old ad agency has been fired; we should see some new Amiga ads real soon now.

The Future

RJ's advice for A1000 owners: "Keep what you've got. It's not worth it to trade up. The A1000 is really a better machine." This may be sour grapes on RJ's part, since the Amiga 2000 was designed in Braunschweig, West Germany, and the version of the A2000 being worked on in Los Gatos was rejected in favor of the Braunschweig-Commodore version. However the A1000 compares to the A2000, though, the Los Gatos 2000 would have certainly been better than either machine. C= management vetoed it because Braunschweig promised a faster design turnaround (and, to their credit, were much faster in execution than the Los Gatos group would have been) and more cost-reduction, which was their specialty. Los Gatos, on the other hand, wanted a dream machine with vastly expanded capabilities in every facet of the machine. The cruel financial facts forced C= to go with the Business Computer Group, who did the Sidecar in Braunschweig as well, and quickly and cheaply.

So they fired more than half the staff at the original Los Gatos facility, one by one. That trauma was to some extent played out on the net; no doubt many of you remember it as a very difficult and emotional time. There are now only six people left in Los Gatos, and their lease expires in March, so thus expires the original Amiga group. And that's how RJ ended his talk; the rise and fall of Amiga Computer Inc. The future of the Amiga is now in the hands of Westchester and Braunschweig, and who knows what direction it will take?

PART 2 - Technical Questions From the Audience

I'll just make this part a list of technical questions and answers, since that was the format at the talk anyway. This part is part technical inquiries and part total rumor mill; caveat emptor.

Q's are from the audience, A's are =RJ=.

Q: When is 1.3 coming and what's in it?
A: 1.3 (or maybe it'll be called 1.2A) will be mostly just 1.2 with hard disk boot; it'll look for Workbench on dh0: as well as df0:. No one is working on it right now, although there are people in WestChester planning it.

Q: Can you do double buffering with Intuition?
A: Pop answer: No. Thought-out: well, yes, but it's not easy. Use MenuVerify and don't change the display while menus are up. It's pretty hairy.

Q: How big is intuition (source code)?
A: The listings (commented) are about a foot thick, 60 lpp, 1 inch margins.

Q: Where did MetaComCo come into the Amiga story?
A: MCC's AmigaDos was a backup plan; the original Los Gatos-written AmigaDos was done with some co-developers who dropped out due to contract and money hassles when C= bought Amiga. Then MCC had to crank EXTREMELY hard to get their BCPL Dos into the system at the last possible minute.

Q: Why isn't the Sidecar out?
A: Who knows? It passed FCC in December.

Q: Why no MMU?
A: Several reasons. Obviously, cost was a factor. MMUs available at the time the Amiga was designed also consumed system time [this is what he said - I'm just the scribe]; although newer MMUs solve this problem they were too late for the Amiga. Secondly, the original goal of the Amiga was to be a killer game machine with easy low-level access, and an MMU didn't seem necessary for a game machine. Third, [get this!] with an MMU, message-passing becomes MUCH MUCH hairier and slower, since in the Amiga messages are passed by just passing a pointer to someone else's memory. With protection, either public memory would need to be done and system calls issued to allocate it, etc., or the entire message would have to be passed. Yecch. So the lack of MMU actually speeds up the basic operation of the Amiga several fold.

Q: Why no resource tracking?
A: The original AmigaDos/Exec had resource tracking; it's a shame it died.

Q: How is your game coming? [??]
A: It's just now becoming a front-burner project. It's number crunch intensive; hopefully it will even take over the PC part of the 2000 for extra crunch. It's half action, half strategy; the 'creation' part is done, only the playing part needs to be written. Next question.

Q: Will there ever be an advanced version of the chip set?
A: Well, Jay Miner isn't working on anything right now. [RUMOR ALERT] The chip folks left in Los Gatos who are losing their lease in March were at one time thinking about 1k square 2meg chip space 128-color DACs though and even stuff like a blitter per plane (!!) They were supposed to be done now, in the original plans; the chip designers will be gone in March, but the design may (?) continue in Westchester. Maybe they'll be here two years from now.

Q: What will happen to the unused Los Gatos A2000 design?
A: ??????

Q: Should I upgrade from my 1000 to a 2000?
A: Probably not. The 2000 isn't enough better to justify the cost. Unless you need the PC compatibility, RJ advocated staying with the 1000. After all the 2000 doesn't have the nifty garage for the keyboard!! The A1000 keyboard is better built; you can have kickstart on disk; it's smaller and a LOT quieter, [maybe not than the old internal drives!!!] and uses less power; the 2000 has no composite video out, plus the RGB quality is a tad worse. Composite video (PAL or NTSC) is an extra-cost option with the 2000.

Q: Have you ever seen a working Amiga-Live!?
A: Yes, I've seen it taking 32-color images at 16fps, and HAM pictures at something like half that. [!!] It's all done and working. I don't know why it's not out. It sure beats Digiview at 8 seconds per image!

Q: What do you use for Amiga development tools?
A: Dpaint and Infominder, Aztec C, Andy Finkel's Microemacs.

A: What's the future of the A1000?
A: They aren't making any right now; they're just shipping from stock. But they do claim that they intend to continue making them.

Q: Is MetaComCo's stuff all really slow, or what?
A: Yes, it is slow. But don't knock it, it works.

Q: Who is the competition for Amiga right now?
A: The new Macs are so expensive, they're not a threat to the 2000, much less the 1000. Atari's new stuff "doesn't impress me." [that's all he said]

Q: What can I do about lack of Amiga ads, and the quality of the ones that do exist?
A: Write (don't call) Clive Smith in Marketing at Westchester and tell him they need better ads.

Q: Why are the pixels 10% higher than wide?
A: The hardware came out that way, and it would have been a pain to do it any other way due to sync-rate-multiple timing constraints.

Notes

The preceding tome was produced entirely by placing my terminal cable just next to the microwave on high and wiggling it around like !*(&*h51@!s, so don't take any of it too seriously.

An Introduction to Amiga Animation Using BOBS

By Peter Boreham

Introduction

This article is a simple introduction to using one of the Amiga's basic animation objects, the BOB. I have written this with the assumption that the reader has the Rom Kernal Manual (RKM) and has read at least the introduction to the animation section (pages 103-109 in the RKM), and is able to program in C. Before writing any BOB handling routines, you should type in the files Intuall.h and GelTools.c, given in the RKM (page 184 ff). Alternatively, I can supply a copy to any interested people if they aren't already on the public domain disks. GelTools.c contains routines to initialise, make and delete BOBs which considerably simplifies BOB handling. The purpose of this article is to show you how to use these procedures to simplify BOB handling; you should still have the RKM to refer to for detailed descriptions.

Please note that the documentation at the beginning of GelTools.c is not correct. There is no procedure DeleteVSprite or DeleteBob; these procedures were combined into one procedure - DeleteGel.

A BOB is a special type of image which is drawn onto the playfield. The image must be square, but each image has a 'ShadowMask' associated with it. The ShadowMask is a one bitplane image the same size as the user defined image. Wherever the ShadowMask has a bit set, the corresponding bit in the actual image will be drawn onto the playfield. Wherever the ShadowMask bits aren't set, the image won't be drawn and that part of the image will be transparent. Therefore, the image may be essentially any shape you wish.

The image may be of any height or size, and its depth may be from one to the number of bitplanes in the playfield (supposedly! see the section on making BOBs below). A programmer may define many BOBS for a playfield.

A BOB may be moved around the playfield without altering anything it moves over, or it can leave a 'trace' behind (like a paintbrush). You may check for collisions between the BOB and other BOBS, or the BOB and the boundary.

Using BOBS

To manipulate a BOB you require three C structures. The first is the GelsInfo structure (defined in graphics/rastport.h). This structure points to the list of BOBS and VSprites, and contains general information for both. The second is a BOB structure (defined in graphics/gels.h). This contains some information specific to BOBS. The other structure is a VSprite structure. This structure is used to describe VSprites, but it is also used in conjunction with a BOB structure to fully describe a BOB. This makes it easier for the operating system to manage VSprites and BOBS together.

To use BOBS, you must follow the this basic sequence of operations.

```
Initialise the GELS system.
Make a BOB.
Add the BOB to the system GEL list.
     Display the BOB.     /* do as many times as */
```

```
        Manipulate the BOB. /* needed                    */
    Delete the BOB from the system GEL list.
    Remove the BOB.
    Cleanup
```

Initialisation

Before you can use BOBS, you must initialise the GELS system. The GELS system requires dummy head and tail VSprites to be put in the VSprite list (to ease list manipulation for the operating system). This can be done by calling the ReadyGels procedure in GelTools.h. This procedure requires pointers to the RastPort the GELS will be displayed in, and a pointer to a GelsInfo structure. The RastPort is likely to be the RastPort of the screen, or a window in the screen in which the BOBS are to be displyed. The GelsInfo structure should be declared by the programmer and its address passed to the procedure. The ReadyGels procedure allocates and initialises space for the dummy VSprites the system needs. It also allocates space for a collision table, and arrays for sprite position and colour information (though the last two are not used by BOBS).

Making a BOB

Before you can use a BOB, you need to initialise BOB and VSprite structures, and attach them to the system list of VSprites. Most of this work will be done for you if you call the MakeBob routine (in GelTools.h) with the appropriate parameters. These are:

- bitwidth. This shows how many bits wide the BOB image is. However, remember that all image data must be WORD aligned.
- lineheight. This shows how many lines high the BOB image is.
- imagedepth. This gives the number of bitplanes of image you have supplied for the BOB. The RKM says it may be from one up to the number of bitplanes for the current screen. However, I have not been able to make a BOB without the system crashing, unless the BOB has the same number of bitplanes as the playfield on which it is displayed.
- planePick. This value shows which BOB image plane is to be placed into each screen bitplane (see page 138 in the RKM).
- planeOnOff. This tells which value (0 or 1) should be placed into bitplanes not supplied with data (see page 140 & 141 in the RKM).
- x,y. These are the initial co-ordinates you wish to give the BOB.
- flags. These flags are for the VSprite structure associated with the current BOB. They may be

SAVEBACK - This tells the system to save the background behind the BOB, so it can be restored later. Geltools.c automatically allocates space for the system to save the background, although it won't be used if you don't set SAVEBACK (see page 144 in the RKM).

OVERLAY - This tells the system to use the BOBS ImagedShadow mask when drawing the BOB image. The BOB will only be drawn onto the screen where the corresponding bits of the ImageShadow are set (see page 144 in the RKM).

The BOBS own flags are set to NULL by the MakeBob procedure. It assumes the old BOB image will be erased when the BOB is moved, and the BOB is not part of an AnimComp. All the BOB structure flags are shown on pages 145 & 146 in the RKM.

Adding a BOB to the GEL list

Before a BOB will be displayed, it must be added to the system GEL list. This is done by calling the system procedure AddBob. AddBob requires two pointers: the address of the BOB to be added and the address of the RastPort on which the BOB will be displayed.

Displaying a BOB

Displaying a BOB requires two steps. Firstly, the list of BOBS must be sorted by calling the system routine SortGList, passing a pointer to the RastPort on which the BOBS are to be displayed. This gets the list of BOBS in order for drawing. To actually display the BOBS, call the system routine DrawGList. It requires two parameters; the first is a pointer to the RastPort on which the BOBS will be displayed, and the second is the ViewPort in which the BOBS will be displayed. After this routine is called the BOBS will be displayed.

Manipulating a BOB

Two basic BOB characteristics which may be altered are:

- its position. To alter a BOBS position simply change its x,y co-ordinates in its associated VSprite structure.

- its appearance. Altering the appearance of a BOB involves two steps. Firstly, the ImageData pointer in the VSprite structure must be altered to point to the new image. Secondly, you should call the system routine InitMasks to change the Collision and BorderLine masks to reflect to new BOB shape (see pages 139, 156,157 & 158 in the RKM for detailed explanation about these structures).

A BOBS appearance should be changed only by altering its image data (and/or depth), NOT its size. If you want to alter a BOB's size, you should remove it from the BOB list, redefine the attributes you want altered and then add it to the list. See the section 'Traps' for further explanation.

Removing a BOB

When you have finished using a BOB, it needs to be removed from the system list and the background restored. This can be done in two ways:

- you may call the system routine RemBob. This requires you to pass a pointer to the BOB which is to be removed (the RKM is incorrect, the RastPort pointer is not passed). Currently, RemBob is defined as a macro in graphics/gels.h which merely sets the BOBSAWAY flag for that BOB. RemBob removes the BOB image from the display at the next call to DrawGList.

- you may call the system routine RemIBob. This routine requires pointers to the BOB, and the RastPort and ViewPort in which it is displayed. It removes the BOB from the display immediately, and also any BOBs which come after it in the display order. Therefore, you will probably want to call SortGList and DrawGList to redraw any other BOBs which may have been removed as well.

Deleting a BOB

If you used MakeBob to allocate and initialise a BOB structure, the allocated memory must be returned to the system. After removing the BOB from the system list (if you used RemBob to remove it from the system GEL list, make sure you call SortGList and DrawGList to actually remove it from the display) you must call DeleteGel to reallocate the memory assigned to it. You need to pass a pointer to the VSprite associated with the BOB. Don't delete BOB's that weren't initialised using MakeBob, this will most likely cause the system to crash.

Cleanup

After you have finished using the GEL system, you need to deallocate space allocated by ReadyGels. Do this by calling PurgeGels. PurgeGels requires a pointer to a GelsInfo structure to be passed to it.

Traps, crashes and bugs:

Making BOBs

If the system crashes while you are drawing BOBs, it may be because your BOB image has less bitplanes than the playfield on which it is displayed. I cannot make a BOB which won't crash the system unless it's image has the same number of bitplanes as it's playfield.

Adding BOBs

DON'T add the same BOB to the system list more than once. This usually causes the system to crash. However, it doesn't seem to hurt the system if a BOB is removed more than once (though you shouldn't need to do this with good programming).

Changing a BOB's appearance

A BOB's appearance may be changed by pointing the ImageData variable in the associated VSprite structure to differnet image data. However, this image must be the same width and height (though not depth) as the previous image. This is because the VSprite structure doesn't record the width and height of the last image put on the screen; it expects them to remain the same. Therefore, if you change the dimensions of the BOB, the next time it is erased and drawn the background will not be restored correctly. The only way to safely change the dimensions of a BOB is to remove it from the GEL list, alter it and then add it again.

Deleting BOBs

Remember: don't call DeleteGel to delete a BOB not initialised using MakeBob. MakeBob allocates memory for tables, and if you initialised a BOB yourself and this memory was not allocated, Nasty Things will happen (you can't deallocate memory that wasn't allocated in the first place).

Cleaning up

Don't call PurgeGels until all BOBs have been removed and deleted.

Drawing on the background

Because BOBS are actually part of the playfield, there is a trap associated with their use. At some time while you are using BOB's, you will probably want to alter the background on which they are displayed. If there are any BOBs on the area you want to alter, they will have to be removed before the area is altered. If they aren't, the next time DrawGList is called the old background will be replaced where the BOB was, destroying the changes you just made. Therefore, make sure there are no BOB's on any background you wish to alter.

The major problem this introduces is flicker. If you need to alter the background often, you will see the BOBs flicker because of the wait imposed while you draw onto the background. However, if you only take a short time to redraw the background you shouldn't have much trouble.

Further topics

This article is a very simple introduction to using BOBs. From my experience, the hardest part is to get a simple program working in the first place! Further BOB topics you can explore are collision detection and double buffering, both of which are covered well in the RKM (also see the example program in the RKM). I hope the introduction and tips I have given here will be of use to anyone starting to use BOBs. Please contact me if you have any further questions or information about BOB usage.

Sample Program:

```
/*************************************************
            Simple BOB program:
   Make a Lores screen, then put on four BOBs and bounce them around the
screen.  This is a very simple example to show the basics of using BOBs, if
you don't want to start with the more complicated example given in the RKM.
 *************************************************/

#include "intuition/intuall.h" /* on page 184 of the RKM */

/* Image Data */

#define IWIDTH  16
#define IHEIGHT 12

WORD PumpkinImage[2][24] = {
    0x0080, 0x3D3C, 0x4342, 0x4182, 0x8C31, 0x8001, 0x9009, 0x8811,
    0x47E2, 0x4002, 0x2004, 0x1FF8,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000,

    0x0000, 0x0000, 0x0000, 0x0000, 0x0080, 0x1D38, 0x2344, 0x4182,
    0x4C32, 0x8001, 0x9FF9, 0xFFFF,
    0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
    0x0000, 0x0000, 0x0000, 0x0000
    };
```

```
#define SWIDTH  320
#define SHEIGHT 200
#define SDEPTH  2
struct Screen *MyScreen;
struct NewScreen NS = { 0,0, SWIDTH,SHEIGHT,SDEPTH, 0,1, NULL,
     CUSTOMSCREEN,
                        NULL,NULL,NULL,NULL };

int Error;
#define OK  0

struct RastPort *GRP;
struct ViewPort *GVP;

struct GelsInfo MyGInfo;
struct Bob *MyBob[4];

SHORT BobVX[4] = {-3,1,-2,2};
SHORT BobVY[4];
SHORT SquashState[4];

LONG    IntuitionBase,GfxBase;

    /* Open libraries and initialise some global pointers */
    int Init()
        {
        if (!(IntuitionBase = OpenLibrary("intuition.library",0)))
            return(Error = 1);
        if (!(GfxBase = OpenLibrary("graphics.library",0)))
            return(Error = 2);
        if (!(MyScreen = (struct Screen *)OpenScreen(&NS)))
            return(Error = 3);
        GRP = &MyScreen->RastPort;
        GVP = &MyScreen->ViewPort;
        return(OK);
        }

    /* Initialise the BOB's and add them to the system list */
    int SetUpGels()
        {
        int Loop;
        static SHORT StartX[4] = { 5,50,100,150 };
        static SHORT StartY[4] = { 10,100,50,150 };
        static SHORT PP[4]     = { 0x02,0x01,0x02,0x01 };

        if (ReadyGels(&MyGInfo,GRP))
            return(Error = 5);
        for (Loop = 0; Loop < 4; Loop++)
            {
            if (!(MyBob[Loop] = (struct Bob *)MakeBob(
                      IWIDTH,IHEIGHT,2,PumpkinImage[0],PP[Loop],0x00,
                      StartX[Loop],StartY[Loop],SAVEBACK | OVERLAY )))
                return(Error = 4);
            MyBob[Loop]->BobVSprite->MeMask = 0;
            AddBob(MyBob[Loop],GRP);
            }
        return(OK);
        }
```

```
/* Move the BOBs around.  You could use collision detection to reverse
  the the direction of the BOBs */
MoveGels()
    {
    static int SquashImage[] = { 0,1,0 };
    int Loop,Time;
    struct VSprite *VS;

    for (Time = 0; Time < 503; Time++)
        {
        Delay(1); /* Slow it down a little */
        for (Loop = 0; Loop < 4; Loop++)
            {
            VS = MyBob[Loop]->BobVSprite;
            if (SquashState[Loop])
                {
                VS->ImageData = /* alter image for the bounce */
                        PumpkinImage[SquashImage[--SquashState[Loop]]];
                InitMasks(VS); /* make new masks so it draws correctly */
                }
            else
                {
                VS->Y = VS->Y + BobVY[Loop];
                if (VS->X <= 0 | VS->X >= SWIDTH-IWIDTH)
                    BobVX[Loop] = -BobVX[Loop];
                VS->X = VS->X + BobVX[Loop];
                if (VS->Y >= SHEIGHT-IHEIGHT)
                    {
                    VS->Y = SHEIGHT-IHEIGHT;
                    SquashState[Loop] = 3;
                    BobVY[Loop] = -BobVY[Loop]; /* reverse it */
                    }
                else
                    BobVY[Loop]++; /* accelerate it downwards */
                }
            }
        SortGList(GRP);
        DrawGList(GRP,GVP); /* now redraw them in their new positions */
        }
    }

/* Clean up everything */
CloseAll()
    {
    int Loop;

    for (Loop = 0; Loop < 4; Loop++)
        if (MyBob[Loop])
            RemBob(MyBob[Loop]->BobVSprite);
    SortGList(GRP);
    DrawGList(GRP,GVP); /* now erase them */
    for (Loop = 0; Loop < 4; Loop++)
        if (MyBob[Loop])
            if (MyBob[Loop]->BobVSprite)
                DeleteGel(MyBob[Loop]->BobVSprite);
    PurgeGels(&MyGInfo);
```

```
        if (MyScreen)
            CloseScreen(MyScreen);
        if (GfxBase)
            CloseLibrary(GfxBase);
        if (IntuitionBase)
            CloseLibrary(IntuitionBase);
        if (Error)
            exit(Error);
        }

main()
    {
    if (Init())
        CloseAll();
    if (SetUpGels())
        CloseAll();
    MoveGels();
    CloseAll();
    }
```

CHEAP Disks & Disk Storage Boxes

Both Nashua and Xidex disks are still available to members at our monthly meetings. Boxes of 10 double sided disks of either brand sell for $ 38 each. Lockable disk storage boxes are now also available. These will hold 80 3½inch disks and cost an incredibly low $ 16.

If you can't make it along to one of the meetings these items are also available from Jonathon Bishop at ANU during the week. Contact him on 493786.

The Amiga Environment Part [illegible] Devices
by Craig Fisher

The Input Device

This device is actually a conglomeration of three other devices enabling programs that need inputs from multiple sources to deal with only one device. The input device combines input events from the keyboard (via the Keyboard device), the mouse, (Gameport device) and the timer (Timer device) into a single stream of messages. The console device actually uses the input device to obtain events which it filters and then passes on to programs reading the console.

The input device has only one unit which is shared by all processes wanting to access it.

The one concept fundamental to this device is the idea of an input event stream. This is basically a linked list of structures, each representing an individual event (mouse, timer or keystroke). When the input device receives a message from one of the three devices it monitors it links the event to the end of the input stream. Events in the input stream can only be read by a process called a handler. The input device is used by programs to add a handler to the stream or write new (dummy) events to the input event stream. The input stream can best be thought of as a conveyor belt carrying input messages. Handlers sit at various intervals along the conveyor and may remove a message from it, modify a message or place a new message on the conveyor. What one handler sees on the conveyor depends on what the previous handlers have done. Usually a handler will be looking for certain events on the stream - such as a special key combination for a pop-up program - and when it finds one remove it from the stream and signal its main program to perform some action. A handler can be placed on the input stream by using the **IND_ADDHANDLER** command, supplying a pointer to code in the associated InputRequestBlock. Once a program has finished with the handler it is removed using the **IND_REMHANDLER** command.

To write a new (dummy) event onto the end of the input event stream, the **IND_WRITEEVENT** command is used. A pointer to an initialised input event structure must be supplied in the InputRequestBlock.

The input device also provides two commands to modify the way keys behave. One is for setting the time to elapse before a key repeats (**IND_SETTHRESH**) and the other sets the time between key repeat events (**IND_SETPERIOD**).

There are also commands to let you specify some things about the gameports - which port the mouse is connected to, set the type of device plugged into the mouse port.

Keyboard Device

The keyboard program is one that would very rarely be used by any application programs but a brief description of what it does may be of interest. As there is only one keyboard, there is only one unit of this device also.

Normally the keyboard simply collects keystrokes from the keyboard hardware (register) and passes it on to the input device in the form of an input event structure.

One particularly interesting thing that this device enables a program to do is to add a routine which will be called when the Amiga is reset by pressing [Ctrl][L-Amiga][R-Amiga]. There is actually a chain of 'resethandlers' which are called when the reset combination is pressed. Adding a routine to the reset handler chain via the **KBD_ADDRESETHANDLER** command will cause your routine to also be called when the reset occurs. A program can also remove its reset handler by using the **KBD_REMRESETHANDLER** command. After a reset handler has been called and finished its processing it must send a **KBD_RESETHANDLERDONE** command to the keyboard device to tell it to go on to the next reset handler in the chain.

There is one command that will let you read a keyboard event, **KBD_READEVENT** but this would normally be done by the input device and you would receive your keystroke information through that.

The last keyboard device command is **KBD_READMATRIX** which lets you determine the current state (up or down) of each key in the keyboard matrix. This fills a (16 byte) data block which you supply with 128 bits representing the state of each key.

Gameport Device

The gameport device is similar to the keyboard device in that it is a low level device accessing the hardware directly and is not normally used by applications as its (mouse) events are passed on to the input device. There are two units of the gameport device, one for each of the two gameports (at the right of the machine). These ports are designed to accept various input controllers: a mouse (or mice), a trackball (a mouse turned upside down), a joystick (either directional or proportional) and a light pen.

Like other non-shared devices a unit of the gameport device must be obtained for exclusive use by your program by a call to OpenDevice(). When finished with the port it is returned by calling CloseDevice. For example:

```
struct IOStdReq *game_io_msg;
:
:
error = OpenDevice("gameport.device",1,game_io_msg,0);
:
: /* Set up and then repeatedly read the device */
:
CloseDevice(game_io_msg);
```

This device has five device specific commands that it responds to which I will just describe briefly:

GPD_SETCTYPE - This command allows you to tell the gameport device what sort of controller is to be connected to either of the ports (units). Port 1 (the front one) is equivalent to unit 0 of this device; port 2 is unit 1. The only controller types that this device supports at present are: mouse, absolute joystick, relative joystick, or "no controller". This means that if you wanted to use a light pen or proportional joystick you would have to write your own driver for it. At present I know of no existing software which supports these controllers.

GPD_ASKCTYPE - Using this command you can identify what type of controller is presently connected to the unit of the device you are using (or at least what the gameport device has been told is there). Note that there is a mistake on page 347 of the Libraries and Devices manual where this command is referred to as **GPD_GETCTYPE.**

GPD_SETTRIGGER - This command is used to tell the gameport device what conditions are to be met before a gameport event is generated. You can tell it to generate an event after a given amount of time has elapsed, when a button is pushed or released or the mouse is moved by a minimum X or Y amount. When a read of the gameport device is attempted a response will not be received until an event is triggered.

GPD_ASKTRIGGER - This command can be used to determine what conditions are currently set to trigger a gameport device event.

GPD_READEVENT - Once the gameport unit has been initialised appropriately you can read its status with this command. If you have set the trigger conditions for a gameport event then your read request will not be replied to until the given conditions are met. A completed input event structure is then returned which can be examined to determine what happenedand the current status of the device. The event may have been the left or right mouse button being pressed or released, or a mouse movement or just a status report after a timeout.

It should be noted that the input device usually has a hold on the front gameport (unit 0) and is therefore unavailable. This means you must either use only the rear port (unit 1), tell the input device to use the rear port (using the input device's **SETMPORT** command) or somehow disable the input device (which means disabling both the console device and intuition). Trying to use the front gameport at the same time as the input device will probably result in you getting an input event each or something else equally unsuccessful.

The Narrator Device

This device embodies one of the Amiga's most notable features - its ability to produce quite high quality speech. This device is in fact used in conjunction with the translator library which provides the single function Translate() to convert a text string into a string of phonemes. The phoneme string is then passed to the narrator device to be reproduced as speech.

This device is not a ROM resident device but a disk resident one - in order to use it the file Narrator.Device must exist in the directory that DEVS: is assigned to when the OpenDevice is attempted. If you intend to use the Translator library (the Narrator can be used without it) the the translator.library file must be in the directory which LIBS: is assigned to when the OpenLibrary call is executed.

The narrator device is, surprisingly, a two way device. Writing to it you can produce speech of a certain pitch, gender, speed, volume etc. Reading from the device you can obtain values representing the shape of a mouth saying the sounds so you can create a graphic face speaking in time with the narrator device.

The Libraries and Devices RKM, Narrator Device chapter contains a fairly extensive tutorial on creating speech with phonemes. This should be consulted if you intend to use speech intensively in your application.

The narrator device uses an extended form of IOStdReq structure which has various other fields tacked onto the end of it. The structure is:

```
struct narrator_rb {
     struct IOStdReq message; /* Standard IO Request Block */
     UWORD rate;              /* words/minute */
     UWORD pitch;             /* baseline pitch in Hertz */
     UWORD mode;              /* monotone or expressive */
     UWORD sex;               /* of voice */
     UBYTE *ch_masks;         /* pointer to audio alloc maps */
     UWORD nm_masks;          /* number of audio alloc maps */
     UWORD volume;            /* from 0 (off) to 64 */
     UWORD sampfreq;          /* audio sampling frequency */
     UBYTE mouths;            /* generate mouths if not 0 */
     UBYTE chanmask;          /* internal device use */
     UBYTE numchan;           /* internal device use */
     UBYTE pad;               /* for alignment */
};
```

Most of these fields are pretty obvious or needn't be touched.
rate - lets you specify how fast the speech will come out.
pitch - the mode is expressive then the pitch will vary above and below this base value.
mode - this is set to either ROBOTICFO (monotone voice) or NATURALFO (expressive voice).
sex - set this to either MALE or FEMALE.
volume - set to 0..64 as with the audio device.

This structure is initialised to default values when the narrator device is opened.

To get the narrator to speak your string of phonemes you set the data pointer of the IOStdReq structure to point to the start of the string and the length field of that structure must be set to the length of the string. For example:

```
struct *writeNarrator;
char outstr[80];
:
:
writeNarrator->message.IO_Data = (APTR)outstr;
writeNarrator->message.IO_Length = strlen(outstr);
:
:
SendIO(writeNarrator);   /* tell narrator to say string */
```

There is another structure associated with the narrator device which is used to read the mouth values as sounds are spoken. This structure is:

```
struct mouth_rb {
     struct narrator_rb voice;     /* speech IO req block */
     UBYTE width;                  /* of mouth */
     UBYTE height;                 /* of mouth */
     UBYTE shape;                  /* device internal use */
     UBYTE pad;                    /* for alignment */
};
```

This structure is passed to the narrator device as a CMD_READ command to obtain mouth width and height values to match the sounds being spoken so a graphical face could be drawn to appear to be speaking. The public domain program SmoothTalker utilizes this and

Programming In C

by Jonathan Bishop

This month we consider some more C programming style issues. In the first issue of the magazine we discussed how to break your C program into a series of modules that make the 'housekeeping' associated with large programs easier. Now we look at how to write more readable and hence debuggable C code.

Throughout this article I have included portions of a module designed to enhance the system Exec list structure to be more powerful than that provided by the Exec library. Unfortunately the full modules are too big to be included in this article in their entirety. We have decided to prepare a disk/series of disks that will be available (free + copy charge) containing code that appears in these articles. The full modules (there are actually two of them) will thus shortly be available on disk from our Software Librarians: Simon Tow and John Perkins.

Perhaps the first impression that a beginning C coder has is that C is *very* difficult to read. This does not have to be so, and in fact good C code should be easy to read and flow smoothly. There is little of merit in being able to write unreadable code, just as there is no great skill required to write unreadable prose! Program code is a method of communicating an algorithm. The emphasis here, should be on communication.

Clear code means fewer bugs, and faster debugging.

FORMATTING

The first way to write clear code is to use sensible formatting. Good formatting is also one of the best ways to pevent bugs, such as those caused by incorrect bracketing. For example compare this:

```
SimpleExample( x, y)
char *x;
{
     int MyLocalVar;

     printf("%s\n", "This is a simple example.");
     if( y>5 )
     {
          printf("This is the first nest.\n");
          if( y<10 )
               printf("This is the second nest.\n");
          else
          {
               int MySecondLocalVar = 1;

               y=MySecondVar;
               printf("This is the third nesting.\n");
          }
     }
     else
         return y;
}
```

with this:

```
SimpleExample( x, y)
char *x;
{
     int MyLocalVar;

     printf("%s\n", "This is a simple example.");
     if( y>5 ){
          printf("This is the first nest.\n");
     if( y<10 )printf("This is the second nest.\n");
     else{
          int MySecondLocalVar = 1;

          y=MySecondVar;
          printf("This is the third nesting.\n");
          }
     }
     else
          return y;
}
```

The points to note are the way the brackets are positioned and the indenting style for variable declarations and if - else statements.

MACROS

One of the most powerful features of the C language is the macro facility provided by the preprocessor. Macros are a text substitution tool. When we define a macro we are instructing the preprocessor to replace every occurrence of the macro string it sees in the file with the string on the right hand side of the macro definition.

Macros can be used to serve some very useful purposes. The following are good examples of using macros to reduce the incidence of some common bugs.

```
#define is ==
#define isnt !=

#define Addr( x ) (&x)

#define Set(x,y) (x = y)
```

The macros 'is' and 'isnt' prevent a common error that comes from dropping an '=' sign in the '==' equivalence check and accidentally assigning.

When we wish to pass a pointer to a structure or variable we do so by taking the address of it. This is achieved by preceding the variable with the '&' symbol. However this can look messy and does not add to clarity so the Addr macro is used instead.

Note that this macro and most others that I will be using are designed to be used where ever you would use a function call. To that end they do not contain semicolons and are surrounded with brackets '()' to ensure that the precedence of the operators is not interfered with by the places in which the macros are used.

Declaring Structure Typedefs

Another extremely useful feature of the C language is the 'typedef' facility. Typedef allows you to declare your own types that you may then treat in the same way as the primitive types. The primitives provided include:

integer, long, short, double, float, char.

We shall consider typedef as used in a structure declaration. There are two parts to this declaration. The actual structure being declared and the macros that are defined to access it. The structure being declared is a List node intended to serve as the work horse for a more powerful set of List handling routines. The actual design of the node is the same as that used by the exec, except that I have added a little bit and made the access to it easier (with macros). For all intents and purposes it seems exactly the same to the exec routines as the List node structure that the exec uses.

```
/*****************************************************
 LNODE TYPE
  List member node.
*****************************************************/
/*************
   Initialise
*************/

#define NodeInit( nd ) ((*((LNODEPTR) nd)) = 0)

/*************
   LookUp Fields in the LNODE structure.
*************/

#define NodeType( nd ) (((LNODEPTR) nd)->Nodes.Is_Node.In_Type)
#define NodePrio( nd ) (((LNODEPTR) nd)->Nodes.Is_Node.In_Pri)
#define NodeName( nd ) (((LNODEPTR) nd)->Nodes.Is_Node.In_Name)
#define NodeSucc( nd ) (((LNODEPTR) nd)->Nodes.Is_LNode.Succ)
#define NodePred( nd ) (((LNODEPTR) nd)->Nodes.Is_LNode.Pred)
#define NodeData( nd ) (((LNODEPTR) nd)->Is_Data)


/************
 Set Fields in the LNODE structure.
************/

#define SNodeType(nd,typ) Set(nd->Nodes.Is_Node.In_Type,       \
                                        ((UBYTE) typ))
#define SNodePrio( nd, prio ) Set( nd->Nodes.Is_Node.In_Pri, \
                                        ((BYTE) prio))
#define SNodeName( nd, name) Set( nd->Nodes.Is_Node.In_Name, \
                                        ((char *) name))
#define SNodeData( nd,data ) Set(nd->Is_Data, ((APTR) data))
#define SNodeSucc( nd, suc ) Set(nd->Nodes.Is_LNode.Succ,      \
                                        ((LNODEPTR)suc))
#define SNodePred( nd,pred ) Set(nd->Nodes.Is_LNode.Pred,      \
                                        ((LNODEPTR)pred))

/*************
 lnode data structure
*************/
```

```
/* None of the AMIGADOS data structures are declared with
typedefs -- I dont know why -- so these two declarations simply
declare two types so that I can refer to them nicely later...*/

typedef struct Node *NODEPTR;
typedef struct Node NODE;

/* The Sub node is my package intended to mirror the NODE
structure only providing TYPE compatibility to point to my LNODE
structure without the compiler screaming. Hence the latter union with
the LSUBNODE and the LNODE.  Since I am only interested in the
pointers here, the rest of the data normally present in a NODE
structure is merely padded here.  This is not strictly necessary but
is done purely from habit. The Compiler should work out the sizes
correctly and align the first fields of both structures (NODE and
LSUBNODE) correctly anyway, but why take chances ? */

typedef struct lsubnode
{
  struct lnode *Succ, *Pred;
  UBYTE Pad1;
  BYTE  Pad2;
  char *Pad3;
} LSUBNODE, *LSUBNODEPTR;

/* This is the actual List node. Basically it is just the Exec
list structure with data field added. The LSUBNODE in the union
is simply sugar to stop type compatibility warnings from the
compiler. */

typedef struct lnode
{
  union                    /* The promised union of nodes */
  {
     NODE Is_Node;         /* This is the same as theirs */
     LSUBNODE Is_LNode;    /* This one is mine.          */
  } Nodes;
  APTR Is_Data;           /* This is the data I carry (any) */
} LNODE, *LNODEPTR;
```

Notice that the structure has a number of macros declared with it. These allow us to produce code free of sidearrows and structure dereferences which can make for very hard reading. A further advantage is that if you change aspects of a structure field, only the macro need be changed and the change is automatically reflected in all the references to that field using the macro.

This technique of setting up structures in C saves me hours in the debugging phase.

Certain conventions are followed in the typedef'ing of a structure. These are that:

1) the tag field is always in lower case (in the last structure above that is 'lnode').

2) the structure type is declared as the tag field in capitals (eg: 'LNODE').

3) a pointer to the structure is also declared as a type by capitalising the tag field and adding 'PTR' (eg: 'LNODEPTR').

Following are two examples of code using these macros and structures, and another (LISTHEADPTR) which is not described in this article. However the full module will be available on disk from the Librarians as promised.

```
/************************************************
 Name  : NewLNode
 Interf:
 (ln:LNODEPTR,type:UBYTE.prio:BYTE,name:(char *),data:APTR)
 (Example Call)

 UBYTE mylnodetyp;
 LNODEPTR  mynode;
 BYTE myprio;
 char *myname;
 APTR mydata;
 mynode = NewLNode(NULL,mytyp,myprio,myname,mydata);

Preconditions
 Input :
  ln - old listnode to be refreshed, NULL if allocation
   required.
  type - type of node, NULL if unused.
  prio - priority weighting of node, NULL if unused.
  name - ptr to a NULL terminated char string being the name
   of the node.
  data - ptr to block of data for this node to carry.
 Global:  none

Postconditions
Output: NULL if memory allocation error else pointer to a new
 LNode. The LNode is initialised to empty. The Succ and Pred ptrs
  are unaltered.
 Sid.Ef: Decreases free memory pool if ln is NULL. May change
  the contents of a list since the succ & pred ptrs are
  unaltered if an old node is modified.

 Purpose
  Creates an instance of a  LNode data structure and sets it
  to be the values passed as params. The address of this
  structure is returned, NULL if memory allocation error.

 Uses  :
  calloc()
 B R W :
  The correct use of this routine when modifying an old node,
  is to first remove it from its list, alter it, and replace it
  on the list.
 Hist  :
  07Jul87 JGB Entered routine.
************************************************/
```

```
LNODEPTR NewLNode(ln,type,prio,name,data)
LNODEPTR ln;
UBYTE type;
BYTE prio;
char *name;
APTR data;
{
  char *calloc();

  if((ln)?ln:(ln=(LNODEPTR)calloc(1,sizeof(LNODEPTR))))
  {
     SNodeType( ln, type ) ;
     SNodePrio( ln, prio );
     SNodeName( ln, name);
     SNodeData( ln, data );

     return ln;
  }
  return NULL;
}

/*****************************************************
 Name  : FindList
 Interf:
   (lh:LISTHDPTR, nd:LNODEPTR, direction:int, data:APTR):LNODEPTR
 (Example Call)
 LISTHDPTR lh;
 LNODEPTR nd;
 APTR mydata= data address
  nd=FindList(lh,NULL,NULL,mydata);
Preconditions
 Input :
  lh - ListHead to start search from if nd is NULL else unused.
  nd - node to start search from, or NULL.
  direction - direction of search (NULL = forwards)
       (non NULL = backwards)
  data - Ptr to APTR data block to look for.

 Global:
  lh,nd must be well formed lists, or NULL;
Postconditions
 Output:
  Returns LNODEPTR that has its data field set to data, or
  NULL if not found.
 Sid.Ef:
  none.
 Purpose:
  Finds an instance of an LNode data stucture in the list
  referenced by "lh" or "nd" where the data field of the node
  points to the same address as the "data" parameter.
 Uses  :
  nothing special.
 B R W :
  lh and/or nd must reference well formed list or NULL.
 Hist  :
  8jul87 JGB Entered this routine.
*****************************************************/
```

```
LNODEPTR FindList(lh,nd,direction,data)
LISTHDPTR lh;
LNODEPTR nd;
APTR data;
{
  if((nd)||(lh))
    for(nd=(nd)?nd:((direction)?ListTailPred(lh):ListHead(lh));
                  (direction)?NodePred(nd):NodeSucc(nd) ;
                      nd=(direction)?NodePred(nd):NodeSucc(nd))
                  if(NodeData(nd) is data)
                        return nd;
  return NULL;
}
```

ACT AMIGA BULLETIN BOARD SERVICE

Opus 1.0

```
                 __
The World's First   /  \        ---------->     Welcome to
   BBS Network     /|oo \
   * FidoNet *    (_|  /_)          The ACT Amiga Bulletin Board Service
                   _`@/_ \                  Established 4 Oct 1986
 1801 Nodes       |     | \   \\          This is Line 1 to the BBS *
 World Wide!      | (*) |  \   ))
    ______        |__U__| /  \//   A free BBS for all Commodore Amiga owners
   / FIDO \        _//|| _\   /
  (___1____)      (_/(_|(____/ (jm)    OPUS Zone:   3   300/1200/2400 FDX
                                            Net : 626     BELL and CCITT
            Sysop: Mike Hurst-Meyers        Node: 218      1200/75 CCITT

 Telephone   BBS : (062) 59 1137    International : +61 62 59 1137
           Voice :       58 4055    Mail times:  4:15am to 6.45am

  *  Line 2 is 59 1440 but does not support 300 FDX or 1200/75 CCITT

                Non-Amiga owners welcome to drop in and chat!

Remember: to abort all messages, type Ctrl-C, pause Ctrl-S, Ctrl-Q continue
 Time Limits: Members 45 mins (Max 24 hours 90) Non Members 30 mins (45)
```

Here are some Statistics for the ACT Amiga BBS as at 5/7/87

352 members - 236 have registered and can download, 116 haven't

236 registered members - 152 have Amigas
45 have IBM Clones and
39 have other computers

152 Amiga Owners - Act 77
Nsw 38
Qld 15
Vic 13
Wa 5
O/s 4